2.4.2 OSX

OSX is not Linux, but it is Unix under the hood. You will need a terminal emulator (the built-in Terminal program works, but I like iTerm2. You will also need to set up XCode to get command-line developer tools. The method for doing this seems to vary depending on which version of XCode you have.

You may end up with `c99` pointing at `clang` instead of `gcc`. Most likely the only difference you will see is the details of the error messages. Remember to test with `gcc` on the Zoo.

Other packages can be installed using Homebrew. If you are a Mac person you probably already know more about this than I do.

2.4.3 Windows

What you can do here depends on your version of Windows.

- For Windows 10, you can install Windows Subsystem for Linux. This gives you the ability to type `bash` in a console window and get a full-blown Linux installation. You will need to use the `apt` program to install things like `gcc`. If you use an Ubuntu distribution, it will suggest which packages to install if you type a command it doesn't recognize.
- For other versions of Windows, you can install a virtualization program like VirtualBox or VMware and run Linux inside it.
- You may also be able to develop natively in Windows using Cygwin, but this is probably harder than the other options and may produce surprising portability issues when moving your code to Linux.

2.5 How to compile and run programs

See the chapter on how to use the Zoo for details of particular commands. The basic steps are

- Creating the program with a text editor of your choosing. (I like `vim` for long programs and `cat` for very short ones.)
- Compiling it with `gcc`.
- Running it.

If any of these steps fail, the next step is debugging. We'll talk about debugging elsewhere.

2.5.1 Creating the program

Use your favorite text editor. The program file should have a name of the form `foo.c`; the `.c` at the end tells the C compiler the contents are C source code.

Here is a typical C program:

```c
#include <stdio.h>

/* print the numbers from 1 to 10 */

int
main(int argc, char **argv)
{
    int i;

    puts("Now I will count from 1 to 10");
    for(i = 1; i <= 10; i++) {
        printf("%d\n", i);
    }

    return 0;
}
```

examples/count.c

2.5.2 Compiling and running a program

Here's what happens when I compile and run it on the Zoo:

```
$ c99 -g3 -o count count.c
$ ./count
Now I will count from 1 to 10
1
2
3
4
5
6
7
8
9
10
$
```

The first line is the command to compile the program. The dollar sign is my **prompt**, which is printed by the system to tell me it is waiting for a command. The command calls `gcc` as `c99` with arguments `-g3` (enable maximum debugging info), `-o` (specify executable file name, otherwise defaults to `a.out`), `count` (the actual executable file name), and `count.c` (the source file to compile). This tells `gcc` that we should compile `count.c` to `count` in C99 mode with maximum debugging info included in the executable file.

The second line runs the output file `count`. Calling it `./count` is necessary because by default the shell (the program that interprets what you type) only looks for programs in certain standard system directories. To make it run a program in the current directory, we have to include the directory name.

2.5.3 Some notes on what the program does

Noteworthy features of this program include:

- The `#include <stdio.h>` in line 1. This is standard C boilerplate, and will appear in any program you see that does input or output. The meaning is to tell the compiler to include the text of the file `/usr/include/stdio.h` in your program as if you had typed it there yourself. This particular file contains declarations for the standard I/O library functions like `puts` (put string) and `printf` (print formatted), as used in the program. If you don't put it in, your program may or may not still compile. Do it anyway.
- Line 3 is a comment; its beginning and end is marked by the `/*` and `*/` characters. Comments are ignored by the compiler but can be helpful for other programmers looking at your code (including yourself, after you've forgotten why you wrote something).
- Lines 5 and 6 declare the `main` function. Every C program has to have a `main` function declared in exactly this way—it's what the operating system calls when you execute the program. The `int` on Line 3 says that main returns a value of type `int` (we'll describe this in more detail later in the chapter on functions), and that it takes two arguments: `argc` of type `int`, the number of arguments passed to the program from the command line, and `argv`, of a pointer type that we will get to eventually, which is an array of the arguments (essentially all the words on the command line, including the program name). Note that it would also work to do this as one line (as K&R typically does); the C compiler doesn't care about whitespace, so you can format things however you like, subject to the constraint that consistency will make it easier for people to read your code.
- Everything inside the curly braces is the body of the `main` function. This includes
  - The declaration `int i;`, which says that `i` will be a variable that holds an `int` (see the chapter on Integer Types).
  - Line 10, which prints an informative message using `puts` (discussed in the chapter on input and output.
  - The `for` loop on Lines 11–13, which executes its body for each value of `i` from 1 to 10. We'll explain how `for` loops work later. Note that the body of the loop is enclosed in curly braces just like the body of the `main` function. The only statement in the body is the call to

`printf` on Line 12; this includes a format string that specifies that we want a decimal-formatted integer followed by a newline (the `\n`).
- The `return 0;` on Line 15 tells the operating system that the program worked (the convention in Unix is that 0 means success). If the program didn't work for some reason, we could have returned something else to signal an error.

3 The Linux programming environment

The Zoo runs a Unix-like operating system called Linux. Most people run Unix with a command-line interface provided by a **shell**. Each line typed to the shell tells it what program to run (the first word in the line) and what arguments to give it (remaining words). The interpretation of the arguments is up to the program.

3.1 The shell

When you sign up for an account in the Zoo, you are offered a choice of possible shell programs. The examples below assume you have chosen `bash`, the Bourne-again shell written by the GNU project. Other shells behave similarly for basic commands.

3.1.1 Getting a shell prompt in the Zoo

When you log in to a Zoo node directly, you may not automatically get a shell window. If you use the default login environment (which puts you into the KDE window manager), you need to click on the picture of the display with a shell in from of it in the toolbar at the bottom of the screen. If you run Gnome instead (you can change your startup environment using the popup menu in the login box), you can click on the foot in the middle of the toolbar. Either approach will pop up a terminal emulator from which you can run emacs, gcc, and so forth.

The default login shell in the Zoo is `bash`, and all examples of shell command lines given in these notes will assume `bash`. You can choose a different login shell on the account sign-up page if you want to, but you are probably best off just learning to like `bash`.

3.1.2 The Unix filesystem

Most of what one does with Unix programs is manipulate the filesystem. Unix files are unstructured blobs of data whose names are given by paths

consisting of a sequence of directory names separated by slashes: for example `/home/accts/some-user/cs223/hw1.c`. At any time you are in a current working directory (type `pwd` to find out what it is and `cd new-directory` to change it). You can specify a file below the current working directory by giving just the last part of the pathname. The special directory names `.` and `..` can also be used to refer to the current directory and its parent. So `/home/accts/some-user/cs223/hw1.c` is just `hw1.c` or `./hw1.c` if your current working directory is `/home/accts/some-user/cs223`, `cs223/hw1.c` if your current working directory is `/home/accts/some-user`, and `../cs223/hw1.c` if your current working directory is `/home/accts/some-user/illegal-downloads`.

All Zoo machines share a common filesystem, so any files you create or change on one Zoo machine will show up in the same place on all the others.

3.1.3 Unix command-line programs

Here are some handy Unix commands:

man `man` *program* will show you the on-line documentation (the *man page*) for a program (e.g., try `man man` or `man ls`). Handy if you want to know what a program does. On Linux machines like the ones in the Zoo you can also get information using `info program`, which has an Emacs-like interface.

You can also use `man` *function* to see documentation for standard library functions. The command `man -k` *string* will search for man pages whose titles contain *string*.

Sometimes there is more than one man page with the same name. In this case `man -k` will distingiush them by different manual section numbers, e.g., `printf (1)` (a shell command) vs. `printf (3)` (a library routine). To get a man page from a specific section, use `man` *section name*, e.g. `man 3 printf`.

ls `ls` lists all the files in the current directory. Some useful variants:

- `ls /some/other/dir`; list files in that directory instead.
- `ls -l`; long output format showing modification dates and owners.

mkdir `mkdir dir` will create a new directory in the current directory named `dir`.

rmdir `rmdir dir` deletes a directory. It only works on directories that contain no files.

cd `cd dir` changes the current working directory. With no arguments, `cd` changes back to your home directory.

pwd `pwd` ("print working directory") shows what your current directory is.

mv `mv old-name new-name` changes the name of a file. You can also use this to move files between directories.

cp `cp old-name new-name` makes a copy of a file.

rm `rm file` deletes a file. Deleted files cannot be recovered. Use this command carefully.

chmod `chmod` changes the permissions on a file or directory. See the man page for the full details of how this works. Here are some common `chmod`'s:

- `chmod 644 file`; owner can read or write the file, others can only read it.
- `chmod 600 file`; owner can read or write the file, others can't do anything with it.
- `chmod 755 file`; owner can read, write, or execute the file, others can read or execute it. This is typically used for programs or for directories (where the execute bit has the special meaning of letting somebody find files in the directory).
- `chmod 700 file`; owner can read, write, or execute the file, others can't do anything with it.

`emacs`, `gcc`, `make`, `gdb`, `git` See corresponding sections.

3.1.4 Stopping and interrupting programs

Sometimes you may have a running program that won't die. Aside from costing you the use of your terminal window, this may be annoying to other Zoo users, especially if the process won't die even if you close the terminal window or log out.

There are various control-key combinations you can type at a terminal window to interrupt or stop a running program.

ctrl-C Interrupt the process. Many processes (including any program you write unless you trap SIGINT using the `sigaction` system call) will die instantly when you do this. Some won't.

ctrl-Z Suspend the process. This will leave a stopped process lying around. Type `jobs` to list all your stopped processes, `fg` to restart the last process (or `fg %1` to start process `%1` etc.), `bg` to keep running the stopped process in the background, `kill %1` to kill process `%1` politely, `kill -KILL %1` to kill process `%1` whether it wants to die or not.

ctrl-D Send end-of-file to the process. Useful if you are typing test input to a process that expects to get EOF eventually or writing programs using `cat > program.c` (not really recommmended). For test input, you are often better putting it into a file and using input redirection (`./program < test-input-file`); this way you can redo the test after you fix the bugs it reveals.

ctrl- Quit the process. Sends a SIGQUIT, which asks a process to quit and dump core. Mostly useful if ctrl-C and ctrl-Z don't work.

If you have a runaway process that you can't get rid of otherwise, you can use `ps g` to get a list of all your processes and their process ids. The `kill` command can then be used on the offending process, e.g. `kill -KILL 6666` if your evil process has process id 6666. Sometimes the `killall` command can simplify this procedure, e.g. `killall -KILL evil` kills all process with command name `evil`.

3.1.5 Running your own programs

If you compile your own program, you will need to prefix it with `./` on the command line to tell the shell that you want to run a program in the current directory (called '`.`') instead of one of the standard system directories. So for example, if I've just built a program called `count`, I can run it by typing

```
$ ./count
```

Here the "`$ `" is standing in for whatever your prompt looks like; you should not type it.

Any words after the program name (separated by **whitespace**—spaces and/or tabs) are passed in as arguments to the program. Sometimes you may wish to pass more than one word as a single argument. You can do so by wrapping the argument in single quotes, as in

```
$ ./count 'this is the first argument' 'this is the second argument'
```

3.1.6 Redirecting input and output

Some programs take input from **standard input** (typically the terminal). If you are doing a lot of testing, you will quickly become tired of typing test input at your program. You can tell the shell to **redirect** standard input from a file by putting the file name after a `<` symbol, like this:

```
$ ./count < huge-input-file
```

A '>' symbol is used to redirect **standard output**, in case you don't want to read it as it flies by on your screen:

```
$ ./count < huge-input-file > huger-output-file
```

A useful file for both input and output is the special file `/dev/null`. As input, it looks like an empty file. As output, it eats any characters sent to it:

```
$ ./sensory-deprivation-experiment < /dev/null > /dev/null
```

You can also **pipe** programs together, connecting the output of one to the input of the next. Good programs to put at the end of a pipe are `head` (eats all but the

first ten lines), `tail` (eats all but the last ten lines), `more` (lets you page through the output by hitting the space bar, and `tee` (shows you the output but also saves a copy to a file). A typical command might be something like `./spew | more` or `./slow-but-boring | tee boring-output`. Pipes can consist of a long train of programs, each of which processes the output of the previous one and supplies the input to the next. A typical case might be:

```
$ ./do-many-experiments | sort | uniq -c | sort -nr
```

which, if `./do-many-experiments` gives the output of one experiment on each line, produces a list of distinct experimental outputs sorted by decreasing frequency. Pipes like this can often substitute for hours of real programming.

3.2 Text editors

To write your programs, you will need to use a text editor, preferably one that knows enough about C to provide tools like automatic indentation and syntax highlighting. There are three reasonable choices for this in the Zoo: `kate`, `emacs`, and `vim` (which can also be run as `vi`). Kate is a GUI-style editor that comes with the KDE window system; it plays nicely with the mouse, but Kate skills will not translate well into other environements. Emacs and Vi have been the two contenders for the One True Editor since the 1970s—if you learn one (or both) you will be able to use the resulting skills everywhere. My personal preference is to use Vi, but Emacs has the advantage of using the same editing commands as the shell and `gdb` command-line interfaces.

3.2.1 Writing C programs with Emacs

To start Emacs, type `emacs` at the command line. If you are actually sitting at a Zoo node it should put up a new window. If not, Emacs will take over the current window. If you have never used Emacs before, you should immediately type `C-h t` (this means hold down the Control key, type `h`, then type `t` without holding down the Control key). This will pop you into the Emacs built-in tutorial.

3.2.1.1 My favorite Emacs commands

General note: `C-x` means hold down Control and press `x`; `M-x` means hold down Alt (Emacs calls it "Meta") and press `x`. For `M-x` you can also hit Esc and then `x`.

C-h Get help. Everything you could possibly want to know about Emacs is available through this command. Some common versions: `C-h t` puts up the tutorial, `C-h b` lists every command available in the current mode, `C-h k` tells you what a particular sequence of keystrokes does, and `C-h l`

tells you what the last 50 or so characters you typed were (handy if Emacs just garbled your file and you want to know what command to avoid in the future).

C-x u Undo. Undoes the last change you made to the current buffer. Type it again to undo more things. A lifesaver. Note that it can only undo back to the time you first loaded the file into Emacs—if you want to be able to back out of bigger changes, use `git` (described below).

C-x C-s Save. Saves changes to the current buffer out to its file on disk.

C-x C-f Edit a different file.

C-x C-c Quit out of Emacs. This will ask you if you want to save any buffers that have been modified. You probably want to answer yes (`y`) for each one, but you can answer no (`n`) if you changed some file inside Emacs but want to throw the changes away.

C-f Go forward one character.

C-b Go back one character.

C-n Go to the next line.

C-p Go to the previous line.

C-a Go to the beginning of the line.

C-k Kill the rest of the line starting with the current position. Useful Emacs idiom: `C-a C-k`.

C-y "Yank." Get back what you just killed.

TAB Re-indent the current line. In C mode this will indent the line according to Emacs's notion of how C should be indented.

M-x compile Compile a program. This will ask you if you want to save out any unsaved buffers and then run a compile command of your choice (see the section on compiling programs below). The exciting thing about `M-x compile` is that if your program has errors in it, you can type `C-x ‘` to jump to the next error, or at least where `gcc` thinks the next error is.

3.2.2 Using Vi instead of Emacs

If you don't find yourself liking Emacs very much, you might want to try Vim instead. Vim is a vastly enhanced reimplementation of the classic `vi` editor, which I personally find easier to use than Emacs. Type `vimtutor` to run the tutorial.

One annoying feature of Vim is that it is hard to figure out how to quit. If you don't mind losing all of your changes, you can always get out by hitting the Escape key a few times and then typing ~\\\ :qa!\\\ ~

To run Vim, type `vim` or `vim filename` from the command line. Or you can use the graphical version `gvim`, which pops up its own window.

Vim is a *modal* editor, meaning that at any time you are in one of several modes (normal mode, insert mode, replace mode, operator-pending mode, etc.), and the interpretation of keystrokes depends on which mode you are in. So typing

`jjjj` in normal mode moves the cursor down four lines, while typing `jjjj` in insert mode inserts the string `jjjj` at the current position. Most of the time you will be in either normal mode or insert mode. There is also a command mode entered by hitting `:` that lets you type longer commands, similar to the Unix command-line or M-x in Emacs.

3.2.2.1 My favorite Vim commands

3.2.2.1.1 Normal mode

:h Get help. (Hit Enter at the end of any command that starts with a colon.)

Escape
: Get out of whatever strange mode you are in and go back to normal mode. You will need to use this whenever you are done typing code and want to get back to typing commands.

i Enter insert mode. You will need to do this to type anything. The command `a` also enters insert mode, but puts new text after the current cursor position instead of before it.

u Undo. Undoes the last change you made to the current buffer. Type it again to undo more things. If you undid something by mistake, c-`R` (control `R`) will redo the last undo (and can also be repeated).

:w Write the current file to disk. Use `:w filename` to write it to `filename`. Use `:wa` to write all files that you have modified. The command `ZZ` does the same thing without having to hit Enter at the end.

:e filename Edit a different file.

:q Quit. Vi will refuse to do this if you have unwritten files. See `:wa` for how to fix this, or use `:q!` if you want to throw away your changes and quit anyway. The shortcuts `:x` and `:wq` do a write of the current file followed by quitting.

h, j, k, l Move the cursor left, down, up, or right. You can also use the arrow keys (in both normal mode and insert mode).

x Delete the current character.

D Delete to end of line.

dd Delete all of the current line. This is a special case of a more general `d` command. If you precede it with a number, you can delete multiple lines: `5dd` deletes the next 5 lines. If you replace the second `d` with a motion command, you delete until wherever you land: `d$` deletes to end of line (`D` is faster), `dj` deletes this line and the line after it, `d%` deletes the next matching group of parentheses/braces/brackets and whatever is between them, `dG` deletes to end of file—there are many possibilities. All of these save what you deleted into register `""` so you can get them back with `p`.

yy Like `dd`, but only saves the line to register `""` and doesn't delete it. (Think *copy*). All the variants of `dd` work with `yy`: `5yy`, `y$`, `yj`, `y%`, etc.

p Pull whatever is in register `""`. (Think *paste*).

<< and >> Outdent or indent the current line one tab stop.

:make Run `make` in the current directory. You can also give it arguments, e.g., `:make myprog`, `:make test`. Use `:cn` to go to the next error if you get errors.

:! Run a command, e.g., `:! echo hello world` or `:! gdb myprogram`. Returns to Vim when the command exits (control-C can sometimes be helpful if your command isn't exiting when it should). This works best if you ran Vim from a shell window; it doesn't work very well if Vim is running in its own window.

3.2.2.1.2 Insert mode

control-P and control-N These are completion commands that attempt to expand a partial word to something it matches elsewhere in the buffer. So if you are a good person and have named a variable `informativeVariableName` instead of `ivn`, you can avoid having to type the entire word by typing `inf`<control-P> if it's the only word in your buffer that starts with `inf`.

control-O and control-I Jump to the last cursor position before a big move / back to the place you jumped from.

ESC Get out of insert mode!

3.2.2.2 Settings

Unlike Emacs, Vim's default settings are not very good for editing C programs. You can fix this by creating a file called `.vimrc` in your home directory with the following commands:

```
set shiftwidth=4
set autoindent
set backup
set cindent
set hlsearch
set incsearch
set showmatch
set number
syntax on
filetype plugin on
filetype indent on
```

examples/sample.vimrc

(You can download this file by clicking on the link.)

In Vim, you can type e.g. `:help backup` to find out what each setting does. Note that because `.vimrc` starts with a `.`, it won't be visible to `ls` unless you use `ls -a` or `ls -A`.

3.3 Compilation tools

3.3.1 The GNU C compiler `gcc`

A C program will typically consist of one or more files whose names end with `.c`. To compile `foo.c`, you can type `gcc foo.c`. Assuming `foo.c` contains no errors egregious enough to be detected by the extremely forgiving C compiler, this will produce a file named `a.out` that you can then execute by typing `./a.out`.

If you want to debug your program using `gdb` or give it a different name, you will need to use a longer command line. Here's one that compiles `foo.c` to `foo` (run it using `./foo`) and includes the information that `gdb` needs: `gcc -g3 -o foo foo.c`

If you want to use C99 features, you will need to tell `gcc` to use C99 instead of its own default dialect of C. You can do this either by adding the argument `-std=c99` as in `gcc -std=c99 -o foo foo.c` or by calling `gcc` as `c99` as in `c99 -o foo foo.c`.

By default, gcc doesn't check everything that might be wrong with your program. But if you give it a few extra arguments, it will warn you about many (but not all) potential problems: `c99 -g3 -Wall -pedantic -o foo foo.c`.

3.3.2 Make

For complicated programs involving multiple source files, you are probably better off using `make` than calling `gcc` directly. Make is a "rule-based expert system" that figures out how to compile programs given a little bit of information about their components.

For example, if you have a file called `foo.c`, try typing `make foo` and see what happens.

In general you will probably want to write a `Makefile`, which is named `Makefile` or `makefile` and tells `make` how to compile programs in the same directory. Here's a typical Makefile:

```
# Any line that starts with a sharp is a comment and is ignored
# by Make.

# These lines set variables that control make's default rules.
# We STRONGLY recommend putting "-Wall -pedantic -g3" in your CFLAGS.
CC=gcc
CFLAGS=-std=c99 -Wall -pedantic -g3

# The next line is a dependency line.
# It says that if somebody types "make all"
# make must first make "hello-world".
```

```
# By default the left-hand-side of the first dependency is what you
# get if you just type "make" with no arguments.
all: hello-world

# How do we make hello-world?
# The dependency line says you need to first make hello-world.o
# and hello-library.o
hello-world: hello-world.o hello-library.o
        # Subsequent lines starting with a TAB character give
        # commands to execute.
        # This command uses make built-in variables to avoid
        # retyping (and getting things wrong):
        #  $@ = target hello-world
        #  $^ = dependencies hello-world.o and hello-library.o
        $(CC) $(CFLAGS) -o $@ $^
        # You can put whatever commands you want.
        echo "I just built hello-world!  Hooray!"

# Here we are saying that hello-world.o and hello-library.o
#  should be rebuilt whenever their corresponding source file
#  or hello-library.h changes.
# There are no commands attached to these dependency lines, so
#  make will have to figure out how to do that somewhere else
#  (probably from the builtin .c -> .o rule).
hello-world.o: hello-world.c hello-library.h
hello-library.o: hello-library.c hello-library.h

# Command lines can do more than just build things.  For example,
# "make test" will rebuild hello-world (if necessary) and then run it.
test: hello-world
        ./hello-world

# This lets you type "make clean" and get rid of anything you can
# rebuild.  The $(RM) variable is predefined to "rm -f"
clean:
        $(RM) hello-world *.o
```

examples/usingMake/Makefile

Given a Makefile, make looks at each dependency line and asks: (a) does the target on the left hand side exist, and (b) is it older than the files it depends on. If so, it looks for a set of commands for rebuilding the target, after first rebuilding any of the files it depends on; the commands it runs will be underneath some dependency line where the target appears on the left-hand side. It has built-in rules for doing common tasks like building `.o` files (which contain machine code) from `.c` files (which contain C source code). If you have a fake target like `all` above, it will try to rebuild everything `all` depends on because there is no file

named `all` (one hopes).

3.3.2.1 Make gotchas

Make really really cares that the command lines start with a TAB character. TAB looks like eight spaces in Emacs and other editors, but it isn't the same thing. If you put eight spaces in (or a space and a TAB), Make will get horribly confused and give you an incomprehensible error message about a "missing separator". This misfeature is so scary that I avoided using make for years because I didn't understand what was going on. Don't fall into that trap—make really is good for you, especially if you ever need to recompile a huge program when only a few source files have changed.

If you use GNU Make (on a zoo node), note that beginning with version 3.78, GNU Make prints a message that hints at a possible SPACEs-vs-TAB problem, like this:

```
$ make
Makefile:23:*** missing separator (did you mean TAB instead of 8 spaces?).  Stop.
```

If you need to repair a Makefile that uses spaces, one way of converting leading spaces into TABs is to use the `unexpand` program:

```
$ mv Makefile Makefile.old
$ unexpand Makefile.old > Makefile
```

3.4 Debugging tools

The standard debugger on the Zoo is `gdb`. Also useful is the memory error checker `valgrind`. Below are some notes on debugging in general and using these programs in particular.

3.4.1 Debugging in general

Basic method of all debugging:

1. Know what your program is supposed to do.
2. Detect when it doesn't.
3. Fix it.

A tempting mistake is to skip step 1, and just try randomly tweaking things until the program works. Better is to see what the program is doing internally, so you can see exactly where and when it is going wrong. A second temptation is to attempt to intuit where things are going wrong by staring at the code or the program's output. Avoid this temptation as well: let the computer tell you what it is really doing inside your program instead of guessing.